速度竟然慢了接近 80%！慢了 80%，而我们甚至不能证明这个实现是正确的。

那算完结了吗？没有！我没有基准。基准应该是性能测试的起点，而不是终点。对一个单例的 4000 万次调用能有多快？在没有同步开销的情况下，单线程版本的实现调用 4000 万次单例，为我们提供了一个很好的基准，参见图 9.3。

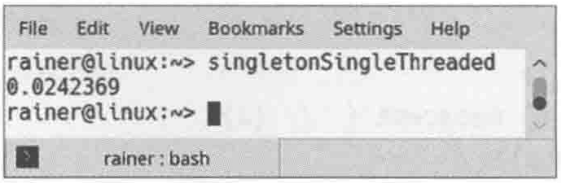


图 9.3 单线程情况下单例的性能

单线程版本执行大约需要 0.024 s，基于 Meyers 单例的多线程版本执行需要 0.035 s。这意味着同步开销使 Meyers 单例慢了 45%。

这种小的同步开销是相当显著的。如果想知道单例模式的线程安全初始化背后的完整故事，请阅读 https://www.modernescpp.com/index.php/thread-safe-initialization-of-a-singleton 上的“Thread-Safe Initialization of a Singleton”（单例的线程安全初始化）一文。文中包含多种其他实现，有基于函数 `std::call_once` 和 `std::once_flag` 的，有基于锁 `std::lock_guard` 的，还有基于利用序列一致（sequential consistency）的原子量的实现。我也提供了在 Linux（GCC）和微软平台（cl.exe）上的性能数据。

## 9.3 启用优化

上一节讨论了错误的假设。现在来看看乐观的方面。

### 使用 Compiler Explorer 获取终极真相

如果想知道哪段代码优化得更好，就必须研究生成的汇编指令。Compiler Explorer 可以为各种编译器生成指令，包括 GCC、Clang 和微软的编译器。并且，有编译器的各个不同版本可供使用。此外，你可以指定编译器的标志，比如，使用 `-O3` 或 `/Ox` 来进行最大优化。

### Per.7 设计应当允许优化

这条规则尤其适用于移动语义，因为你写算法时应该使用移动语义，而不是拷贝语义。移动语义的使用会自动带来一些好处。

1. 算法使用低开销的移动操作，而不是高开销的拷贝操作。
2. 算法要稳定得多，因为它不需要内存分配，所以不会出现 `std::bad_alloc` 异常。
3. 可将算法用于只能移动的类型，如 `std::unique_ptr`。

你可能会在我的论证中发现一个漏洞。当在一个需要移动语义的算法中使用一个“只能拷贝”的类型时，会发生什么？

```
// swap.cpp

#include <algorithm>
#include <iostream>
#include <utility>

template <typename T>
void swap(T& a, T& b) noexcept {  // (2)
    T tmp(std::move(a));
    a = std::move(b);
    b = std::move(tmp);
}

class BigArray {

public:
    explicit BigArray(std::size_t sz): size(sz), data(new int[size]) {}

    BigArray(const BigArray& other): size(other.size),
                                     data(new int[other.size]) {
        std::cout << "Copy constructor" << '\n';
        std::copy(other.data, other.data + size, data);
    }

    BigArray& operator = (const BigArray& other) {
        std::cout << "Copy assignment" << '\n';
        if (this != &other){
            delete [] data;
            data = nullptr;
            size = other.size;
            data = new int[size];
            std::copy(other.data, other.data + size, data);
        }
        return *this;
    }

    ~BigArray() {
        delete[] data;
    }
private:
    std::size_t size;
    int* data;
};
```